Templates
for scalable data analysis

2 Synchronous Templates

Amr Ahmed, Alexander J Smola, Markus Weimer
Yahoo! Research & UC Berkeley & ANU
Running Example

Spam Filter

Inbox

Spam
Running Example

Inbox

Spam

Spam Filter

Monday, April 16, 2012
Running Example
Running Example

- Inbox
- Spam
- Spam Filter
- Webmail UI

Monday, April 16, 2012
Running Example
Running Example

Spam Filter

Inbox

Spam

Logged event

Webmail UI

Monday, April 16, 2012
Running Example

Inbox

Spam

Spam Filter

Logged event

Webmail UI

Monday, April 16, 2012
ML Workflow

- Raw Data
- Example
- Model
ML Workflow

Example Formation

Training

Raw Data Example Model

Log Files Videos Images Text Strokes Geodata Sound

Monday, April 16, 2012
ML Workflow

Example Formation

Training

Prediction

Raw Data

Example

Model

Monday, April 16, 2012
ML Workflow

Example Formation

Prediction

Raw Data
Example
Model

Log Files
Images
Strokes
Geodata

Videos
Text

Sound
ML Workflow

- Sound
- Log Files
- Videos
- Images
- Text
- Strokes
- Geodata

- Example Formation

- Prediction

- Raw Data
- Example
- Model

Monday, April 16, 2012
ML Workflow

- Sound
- Log Files
- Videos
- Images
- Text
- Strokes
- Geodata

Example Formation

Prediction

Raw Data • Example • Model
ML Workflow

Example Formation

Prediction

Raw Data  Example  Model
ML Workflow

Raw Data → Example Formation → Prediction

- Sound
- Log Files
- Videos
- Images
- Text
- Strokes
- Geodata
• Example Formation in Pig
• Modeling today Hadoop, Spark, Pregel
• Future Declarative Systems
Example Formation
Example Formation

EMail

Log Files
Example Formation

EMail → ID → Bag of Words

Log Files
Example Formation

EMail $\rightarrow$ ID $\rightarrow$ Bag of Words

Log Files $\rightarrow$ ID $\rightarrow$ Label
Example Formation

EMail → ID → Bag of Words

Log Files → ID → Label

Label → Bag of Words
Example Formation

EMail → ID → Bag of Words

Log Files → ID → Label

Example
Example Formation

- Feature Extraction
  - Log Files → ID → Label
  - Bag of Words

Example

- Bag of Words
- Label
Example Formation

Feature Extraction

Label Extraction

Example
Requirements

EMail → ID → Bag of Words

Log Files → ID → Label

Data Parallel Functions
Requirements

EMail → ID → Bag of Words

Log Files → ID → Label

Large Scale Join

Label → Bag of Words
Apache Pig

- Relational Query Language
- Similar to SQL
- Performs runtime optimizations
- Executes Queries on Apache Hadoop
- Developed and heavily used by Yahoo!
- Open Source (Apache)

http://pig.apache.org
Pig: Example Formation

• **Feature and Label Extraction**
  • User Defined Function
  • Applied via FOREACH ... GENERATE

• **Example formation**
  • JOIN between the outputs of the above
Machine Learning in MapReduce
MapReduce

- Parallel, Distributed programming framework

- User defines two functions:
  - `map(x)` emits `(key, value)` pairs
  - `reduce(k, x[])` gets all values for a key, produces output
MapReduce
MapReduce

Map
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce

MapReduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
MapReduce

Map

GroupBy (Shuffle)

Reduce
Open Source MapReduce Implementation:

- **HDFS**: Distributed FileSystem
- **YARN**: Resource Management
- **MapReduce**: Programming Framework

http://hadoop.apache.org
• Open Source MapReduce Implementation
  • **HDFS**: Distributed FileSystem
  • **YARN**: Resource Management
  • **MapReduce**: Programming Framework

http://hadoop.apache.org
Efficient Noise-Tolerant Learning from Statistical Queries

MICHAEL KEARNS
AT&T Laboratories—Research, Florham Park, New Jersey

Abstract. In this paper, we study the problem of learning in the presence of classification noise in the probabilistic learning model of Valiant and its variants. In order to identify the class of "robust" learning algorithms in the most general way, we formalize a new but related model of learning from statistical queries. Intuitively, in this model, a learning algorithm is forbidden to examine examples of the unknown target function, but is given access to an oracle providing probabilities over the sample space of random examples.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant's model, with a noise rate approaching the information-theoretic barrier of 1/2. We then demonstrate the generality of the statistical query model, showing that practically every class learnable in Valiant's model and its variants can also be learned in the new model (and thus can be learned in the presence of noise). A notable exception to this statement is the class of parity functions, which we prove is not learnable from statistical queries, and for which no noise-tolerant algorithm is known.

Categories and Subject Descriptors: F. [Theory of Computation]; G.3 [Probability and Statistics]; I.2. [Artificial Intelligence]; I.5 [Pattern Recognition]

General Terms: Computational learning theory, Machine learning

Additional Key Words and Phases: Computational learning theory, machine learning

1. Introduction

In this paper, we study the extension of Valiant's learning model [Valiant 1984] in which the positive or negative classification label provided with each example may be corrupted by random noise. This extension was first examined in the learning theory literature by Angluin and Laird [1988], who form the simplest type of white label noise and then sought algorithms tolerant of highest possible rate of noise. In addition to being the subject of a large number of theoretical studies [Angluin and Laird 1988; Laird 1988; Sloan 1988; Kea...]

• Learning algorithm can access the learning problem only through a statistical query oracle

• The statistical query oracle returns an estimate of the expectation of a function f(x,y) (averaged over the data distribution).
Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to many different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a single algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain “summation form,” which allows them to be easily parallelized on multicore computers. We adapt Google’s map-reduce [7] paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naïve Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

1 Introduction

Frequency scaling on silicon—the ability to drive chips at ever higher clock rates—is beginning to hit a power limit as device geometries shrink due to leakage, and simply because CMOS consumes power every time it changes state [9, 10]. Yet Moore’s law [20], the density of circuits doubling every generation, is projected to last between 10 and 20 more years for silicon based circuits [10].
Example: Gradient

\((x, y)_1\)
\((x, y)_2\)
\((x, y)_3\)
\((x, y)_4\)
Example: Gradient

ComputeGradient

$$(x, y)_1, (x, y)_2, (x, y)_3, (x, y)_4 \rightarrow g_1, g_2, g_3, g_4$$
Example: Gradient

\[ \text{ComputeGradient} \]

\[ \sum \]

\[ g = g_1 + g_2 + g_3 + g_4 \]
Example: Gradient

\begin{align*}
\mathbf{g}(x, y) &= \sum \mathbf{g}(x, y) \\
\mathbf{g}_1 &= \text{ComputeGradient}(x, y_1) \\
\mathbf{g}_2 &= \text{ComputeGradient}(x, y_2) \\
\mathbf{g}_3 &= \text{ComputeGradient}(x, y_3) \\
\mathbf{g}_4 &= \text{ComputeGradient}(x, y_4)
\end{align*}
Example: Gradient

\[
\begin{align*}
&\text{(x, y)}_1 \\
&\text{(x, y)}_2 \\
&\text{(x, y)}_3 \\
&\text{(x, y)}_4 \\
\end{align*}
\]

Map

\[
\begin{align*}
g_1 \\
g_2 \\
g_3 \\
g_4 \\
\end{align*}
\]

Reduce

\[
\sum \\
g
\]

ComputeGradient

Monday, April 16, 2012
• Machine Learning Library
• Implementations of many algorithms, both on Hadoop MapReduce and stand-alone
• Open Source (Apache)
• Welcoming, helpful community

http://mahout.apache.org
• **Recommender Systems**, e.g.
  • User and Item based recommenders
  • Collaborative Filtering
  • Clustering (K-Means, Mean Shift, …)
• **Topic Models** (LDA)
• **Supervised ML**
  • (Logistic) Regression
  • Linear SVMs
  • Decision Trees and Forests
Efficient Noise-Tolerant Learning from Statistical Queries

MICHAEL KEARNS
AT&T Laboratories—Research, Florham Park, New Jersey

Abstract. In this paper, we study the problem of learning in the presence of classification noise in the probabilistic learning model of Valiant and its variants. In order to identify the class of “robust” learning algorithms in the most general way, we formalize a new but related model of learning from statistical queries. Intuitively, in this model, a learning algorithm is forbidden to examine individual examples of the unknown target function, but is given access to an oracle providing estimates of probabilities over the sample space of random examples.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic barrier of 1/2. We then demonstrate the generality of this result by showing that practically every class learnable in Valiant’s model and its variants can also be learned in the new model (and thus can be learned in the presence of noise). A notable exception to this statement is the class of parity functions, which we prove is not learnable from statistical queries, and for which no noise-tolerant algorithm is known.

Categories and Subject Descriptors: F. [Theory of Computation]; G.3 [Probability and Statistics]; I.2. [Artificial Intelligence]; I.5 [Pattern Recognition]

General Terms: Computational learning theory, Machine learning

Additional Key Words and Phases: Computational learning theory, machine learning

1. Introduction

In this paper, we study the extension of Valiant’s model [Valiant 1984] in which the positive or negative classification of an example may be corrupted by random noise. This extension was first studied in the learning theory literature by Angluin and Laird [1988], who formalized the sought algorithms to identify the best of the noisy labels, and then analyzed the highest possible rate of noise. In addition to theoretical studies [Angluin and Laird 1988; Li 1993], the classification noise model has become a common paradigm for experimental machine learning research.
Efficient Noise-Tolerant Learning from Statistical Queries

MICHAEL KEARNS

Abstract. In this paper, we study the problem of learning in the presence of classification noise in the probabilistic learning model of Valiant and its variants. In order to identify the class of “robust” learning algorithms in the most general way, we formalize a new but related model of learning from statistical queries. Intuitively, in this model, a learning algorithm is forbidden to examine individual examples of the unknown target function, but is given access to an oracle providing estimates of probabilities over the sample space of random examples.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise-rate approaching the information-theoretic barrier of 1/2. We then demonstrate the generality of this result by showing that practically every class learnable in Valiant’s model and its variants can also be learned in the new model (and thus can be learned in the presence of noise). A notable exception to this statement is the class of parity functions, which we prove is not learnable from statistical queries, and for which no noise-tolerant algorithm is known.

Categories and Subject Descriptors: F.1.1 [Theory of Computation]; G.3 [Probability and Statistics]; I.2.1 [Artificial Intelligence]; I.5.1 [Pattern Recognition]

General Terms: Computational learning theory, Machine learning

Additional Key Words and Phrases: Computational learning theory, machine learning

1. Introduction

In this paper, we study the extension of Valiant’s learning model [Valiant 1984] in which the positive or negative classification label of an example may be corrupted by random noise. This extension was first studied in [Angluin and Laird 1988; Laird 1988; Sloan 1988; Kea

References


Copyright © 2000 ACM 0004-5411/00/0008-0000 $5.00
Map-Reduce for Machine Learning on Multicore

Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speedup. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to many different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a given algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain “summation form,” which allows them to be easily parallelized on multicore computers. We adapt Google’s map-reduce [7] paradigm to demonstrate this parallel speedup technique on a variety of learning algorithms including locally weighted linear regression (LWR), k-means, logistic regression (LR), naive-Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.
Map-Reduce for Machine Learning on Multicore

Chang-Tao Chu
chentaostanford.edu

Sang Kyun Kim
skkim38@stanford.edu

Yi-An Lin
ianl@stanford.edu

YuanYuan Yu
yuanyuan@stanford.edu

Gary Bradski
garybradski@gmail

Andrew Y. Ng
ang@cs.stanford.edu

Kunle Olukotun
kunle@cs.stanford.edu

Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speedup. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to many different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to parallelize learning algorithms at a time. Specifically, we show that algorithms that fit the Statistical Query model (15) can be written in a certain “summation form,” which allows them to be readily parallelized on multicore computers. We adapt Google’s map-reduce (7) algorithm to demonstrate this parallel speedup technique on a variety of learning algorithms, including linearly weighted least squares (LWLS), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.
Further Reading

Session 2: Modeling with Hadoop
Algorithms in MapReduce

Vijay K Narayanan
Principal Scientist, Yahoo! Labs, Yahoo!

Tutorial @ KDD 2011
http://www.slideshare.net/hadoop
Further Reading

Session 2: Modeling with Hadoop
Algorithms in MapReduce

Vijay K Narayanan
Principal Scientist, Yahoo! Labs, Yahoo!

Tutorial @ KDD 2011
http://www.slideshare.net/hadoop
Further Reading
Further Reading
Efficient Noise-Tolerant Learning from Statistical Queries

MICHAEL KEARN
AT&T Laboratories—Research, Florham Park, New Jersey

1 Introduction

In this paper, we study the problem of learning in the presence of classification noise in the
statistical query model. In particular, we show that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit. One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit. One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit. One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit. One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit.

One of our main results shows that any class of functions learnable from statistical queries is in fact learnable with classification noise in Valiant’s model, with a noise rate approaching the information-theoretic limit.
Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to many different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a single algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain “summation form,” which allows them to be easily parallelized on multicore computers. We adapt Google’s map-reduce [7] paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

1 Introduction

Frequency scaling on silicon—the ability to drive chips at ever higher clock rates—is beginning to hit a power limit as device geometries shrink due to leakage, and simply because CMOS consumes power every time it changes state [9, 10]. Yet Moore’s law [20], the density of circuits doubling every generation, is projected to last between 10 and 20 more years for silicon based circuits [10].
## Abstract

We are at the beginning of the multicore era. Computers will have increasingly many cores (processors), but there is still no good programming framework for these architectures, and thus no simple and unified way for machine learning to take advantage of the potential speed up. In this paper, we develop a broadly applicable parallel programming method, one that is easily applied to *many* different learning algorithms. Our work is in distinct contrast to the tradition in machine learning of designing (often ingenious) ways to speed up a *single* algorithm at a time. Specifically, we show that algorithms that fit the Statistical Query model [15] can be written in a certain “summation form,” which allows them to be easily parallelized on multicore computers. We adapt Google’s map-reduce [7] paradigm to demonstrate this parallel speed up technique on a variety of learning algorithms including locally weighted linear regression (LWLR), k-means, logistic regression (LR), naive Bayes (NB), SVM, ICA, PCA, gaussian discriminant analysis (GDA), EM, and backpropagation (NN). Our experimental results show basically linear speedup with an increasing number of processors.

## Introduction

Frequency scaling on silicon—the ability to drive chips at ever higher clock rates—is beginning to hit a power limit as device geometries shrink due to leakage, and simply because CMOS consumes power every time it changes state [9, 10]. Yet Moore’s law [20], the density of circuits doubling every generation, is projected to last between 10 and 20 more years for silicon based circuits [10].

### Troublemaker

- **ML is iterative**
- **Each iteration is a Job**
- **Overhead per job (>45s)**
  - **Scheduling**
  - **Program Distribution**
  - **Data Loading and Parsing**
  - **State Transfer**
Beyond MapReduce today
Solutions

- Local (subsampling)
- MPI
- Spark
- Pregel
Subsampling

• Form examples on the cluster

• Subsample the data on the cluster

• Train a model on a single machine
Model Averaging

Per-Partition Training

Averaging
Model Averaging

Per-Partition Training → Averaging
Model Averaging

Per-Partition Training → Averaging
Model Averaging

Per-Partition Training

Averaging
• Mature HPC standard

• Supported on many clusters (e.g. OpenMPI)

• Available in C, Fortran and Scripting Languages

• Key operation here: AllReduce
AllReduce
AllReduce

... AllReduce()

... AllReduce()

... AllReduce()
AllReduce
AllReduce

... AllReduce()

... AllReduce()

... AllReduce()

... AllReduce()
AllReduce

... AllReduce()

... AllReduce()

... AllReduce()

... AllReduce()
State Persists Across Iterations
Hadoop AllReduce

- Use Hadoop for
- Data local scheduling
- Good machine identification

- Use MPI for
- AllReduce

- 30x Speedup over Hadoop MapReduce

A Reliable Effective Terascale Linear Learning System

Alekh Agarwal
Department of EECS
UC Berkeley
alekh@cs.berkeley.edu

Miroslav Dudík
Yahoo! Research
New York, NY
mdudik@yahoo-inc.com

Olivier Chapelle
Yahoo! Research
Santa Clara, CA
chap@yahoo-inc.com

John Langford
Yahoo! Research
New York, NY
jl@yahoo-inc.com

ABSTRACT

We present a system and a set of techniques for learning linear predictors with convex losses on terascale datasets, with trillions of features, billions of training examples and millions of parameters in an hour using a cluster of 1000 machines. Individually none of the component techniques is new, but the careful synthesis required to obtain an efficient implementation is a novel contribution. The result is, up to our knowledge, the most scalable and efficient linear learning system reported in the literature. We describe and thoroughly evaluate the components of the system, showing the importance of the various design choices.

1. INTRODUCTION

Distributed machine learning is a research area that has seen a growing body of literature in recent years. Much work focuses on problems of the form

$$\min_{w \in \mathbb{R}^d} \sum_{i=1}^n (l(w^T x_i, y_i) + \lambda R(w)),$$  \hspace{1cm} (1)

where $x_i$ is the feature vector of the $i$-th example, $y_i$ is the label, $w$ is the linear predictor, $l$ is a loss function and $R$ a regularizer. Much of this work exploits the natural decomposability over examples in (1), partitioning the examples over different nodes in a distributed environment such as a cluster.

Perhaps the simplest learning strategy when the number of samples is very large is to subsample a smaller set of examples that can be tractably learned with. However, this strategy only works if the problem is simple enough or the number of parameters is very small. The setting of interest here is when a large number of samples is really needed to learn a good model, and distributed algorithms are a natural choice for such scenarios.

The number of zero entries in the input size divided by the wall clock running time—is

Some prior works (McDonald et al., 2010; Zinkevich et al., 2010) consider online learning with averaging and I et al. (2010a) propose gossip-style message passing algorithms extending the existing literature on distributed convex optimization (Bertsekas and Tsitsiklis, 1989). Lan et al. (2009) analyze a delayed version of distributed online learning. Dekel et al. (2010) consider mini-batch versions of online algorithms which are extended to delay-based up in Agarwal and Duchi (2011). A recent article of Boyd et al. (2011) describes an application of the ADMM technique distributed learning problems. GraphLab (Low et al., 2010) is a parallel computation framework on graphs. More c related to our work is that of Teo et al. (2007) who use it to parallelize a bundle method for optimization.

However, all of the aforementioned approaches see leave something to be desired empirically when deploying large clusters. In particular their throughput—measure the input size divided by the wall clock running time—is smaller than the the I/O interface of a single machine almost all parallel learning algorithms (Bekkerman et al., 2011, Part III, page 8). The I/O interface is an upper bound on the speed of the fastest sequential algorithm since sequential algorithms are limited by the network interfaces in acquiring data. In contrast, we were able to achieve throughput of 500M features/s, which is about a factor faster than the 1Gbps network interface of any one node.

An additional benefit of our system is its compatibility with MapReduce clusters such as Hadoop (unlike MPI-based systems) and minimal additional programming effort to parallelize existing learning algorithms (unlike MapReduce approaches).

One of the key components in our system is a computation infrastructure that efficiently accumulates and broadcasts values across all nodes of a computation. It is functionally similar to MPI AllReduce (hence we use the name) is a reliable and efficient communication library.

http://hunch.net/~vw
MPI: Conclusion

- The Good
  - Computational Performance
  - Well established software available

- The Bad
  - No fault tolerance

- The Ugly
  - Ignorance of shared clusters
  - Systems-Level decisions at the algorithm layer
Spark: Intro

• Open Source cluster computation framework
• Developed at UC Berkeley by the AMP Lab
• Aimed at interactive and iterative use cases
• 30x faster than Hadoop for those
• User interface: Embedded Domain Specific Language in Scala

http://spark-project.org/
val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
    val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

    w -= gradient
}

Monday, April 16, 2012
val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

  w -= gradient
}

Monday, April 16, 2012
Spark: Example

val points = spark.textFile(...).map(parsePoint).partitionBy(HashPartitioner(NODES)).cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)
  w -= gradient
}

Loads data into (distributed) main memory
val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

    w -= gradient
}

Monday, April 16, 2012
val points = spark.textFile(...).
             map(parsePoint).
             partitionBy(HashPartitioner(NODES)).
             cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_, w)).reduce(_ + _)

  w -= gradient
}

Spark: Example
val points = spark.textFile(...).
                   map(parsePoint).
                   partitionBy(HashPartitioner(NODES)).
                   cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)
  w -= gradient
}

Spark: Example
Computes a gradient per data point

Monday, April 16, 2012
val points = spark.textFile(...).
                   map(parsePoint).
                   partitionBy(HashPartitioner(NODES)).
                   cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)
  w -= gradient
}

Computes a gradient per data point
Sums them up
val points = spark.textFile(...).
    map(parsePoint).
    partitionBy(HashPartitioner(NODES)).
    cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
    val gradient = points.map(computeGradient(_,w)).reduce(_ + _)

    w -= gradient
}

Monday, April 16, 2012
val points = spark.textFile(...).
              map(parsePoint).
              partitionBy(HashPartitioner(NODES)).
              cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_,w)).reduce(_ + _)
  
  w -= gradient
}
val points = spark.textFile(...).
map(parsePoint).
partitionBy(HashPartitioner(NODES)).
cache()

var w = Vector.random(D)

for (i <- 1 to ITERATIONS) {
  val gradient = points.map(computeGradient(_, w)).reduce(_ + _)
  w -= gradient
}

Trouble!
(physical layer shows through)
Spark: Conclusion

**The Good**
- Speed (ca. MPI speed)
- Fault Tolerance
- Ease of Programming

**The Bad**
- Main Memory Assumption

**The Ugly**
- Systems aspects creep up
Pregel: A System for Large-Scale Graph Processing

Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski

Google, Inc.
{malewicz,austern,ajcbik,dehnert,ilan,naty,gczaj}@google.com

\*RACT
practical computing problems concern large graphs. Examples include the Web graph and various social networks. The scale of these graphs—in some cases billions of vertices, trillions of edges—poses challenges to their efficient processing. In this paper we present a computational model suitable for this task. Programs are expressed as a sequence of iterations, in each of which a vertex can receive messages sent in the previous iteration, send messages to other vertices, and modify its own state and that of its outgoing edges or mutate graph topology. This vertex-centric approach is flexible enough to express a broad set of algorithms. The model has been designed for efficient, scalable fault-tolerant implementation on clusters of thousands of commodity computers, and its implied synchronicity makes reasoning about programs easier. Distribution-related details are hidden behind an abstract API. The result is a framework for processing large graphs that is expressive, easy to program, and fault-tolerant.

Keywords and Subject Descriptors}
Programming Techniques]: Concurrent Programming—Distributed programming: D.2.13 [Software Engineering]:Reusable Software—Reusable libraries

\*ral Terms
Algorithms

\*ords
Distributed computing, graph algorithms

\*RODUCTION
Internet made the Web graph a popular object of study and research. Web 2.0 fueled interest in social networks. Other large graphs—for example induced by transaction routes, similarity of newspaper articles, paths of disease outbreaks, or citation relationships among papers—have been processed for decades. Frequent applied algorithms include shortest paths computations, frequent itemsets, clustering, and variations on the page rank algorithm. There are many other graph computing problems of practical value, e.g., minimum cut and connected components.

Efficient processing of large graphs is challenging. Algorithms often exhibit poor locality of memory access and may exhibit a changing degree of parallelism over the course of execution. Distribution over machines exacerbates the locality issue, and increases the probability that a machine will fail during computation. Despite the ubiquity of large graphs and their commercial importance, we know of no scalable general-purpose system for implementing arbitrary graph algorithms over arbitrary graph representations in a large-scale distributed environment.

Implementing an algorithm to process a large graph graphically means choosing among the following options:

1. Crafting a custom distributed infrastructure, requiring a substantial implementation effort that must be repeated for each new algorithm or graph representation.

2. Relying on an existing distributed computing platform, often ill-suited for graph processing. MapReduce for example, is a very good fit for a wide array of scale computing problems. It is sometimes used to mine large graphs [11, 30], but this can lead to suboptimal performance and usability issues. The models for processing data have been extended with flavors of clustering, and variations on the page rank—GL [lk] and LEDy [km] or FGL [in], NetworkX [jq], or JDSL [jh], but these extensions are usually not ideal for graph algorithms that often better fit a message passing model.

3. Using a single-computer graph algorithm library as BGL [43], LEDA [35], or Stanford GraphBase [29], or FGL [16], limited scale of problems that can be addressed.

4. Using an existing parallel graph system. The Pregel system is a framework for processing large graphs that is expressive, easy to program, and fault-tolerant.
Giraph

- Apache Open Source implementation of Pregel
- Runs on Hadoop, (ab)uses mappers to do so
- Used at LinkedIn and Facebook

http://incubator.apache.org/giraph/
Pregel Visually

\[ t=0 \]
Pregel Visually

Messages Arrive and Are Processed

t=0
Pregel Visually

Messages Arrive and Are Processed

$t=0$
Pregel Visually

Barrier

<table>
<thead>
<tr>
<th>t=0</th>
<th>t=1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pregel Visually

t=0  t=1
Messages are being sent
Pregel Visually

t=0  t=1

Monday, April 16, 2012
Pregel Visually
Pregel Visually

Termination: No more messages
• `update()` receives the PageRank of all neighbors
• Updates its local PageRank
• Sends new PageRank around if it changed enough
Pregel: Conclusion

• **The Good**
  • Excellent Map for Graph problems
  • Fast

• **The Bad**
  • Memory Model
  • Main Memory Assumption

• **The Ugly**
  • Wrong computational model (stay for the afternoon)
Open Problems

• No complete isolation of user / systems code
  • Unlike MapReduce

• No one system for example formation and modeling
  • Learning Effort
  • Orchestration
  • Wasted resources in distributed clusters
A Declarative Approach
Goals

- Unify Example Formation and Modeling
  - Relational Algebra Operators
  - Iteration Support
  - A unified runtime

- Increase Productivity via high-level language
  - Insulate the user from the systems aspects
  - Debugging and IDE support
Approach
Approach

ScalOps

High Level Language
Relational Algebra and Loops
Approach

- ScalOps
- Relational Algebra and Loops
- Datalog
- Captures the Recursive Dataflow

High Level Language
Declarative Language
Approach

ScalOps

High Level Language
Relational Algebra and Loops

Datalog

Declarative Language
Captures the Recursive Dataflow

Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler
Approach

ScalOps

High Level Language
Relational Algebra and Loops

Datalog

Declarative Language
Captures the Recursive Dataflow

Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler

Hyracks Dataflow

Unified Runtime
Scalability + High performance

Monday, April 16, 2012
Approach

ScalOps

High Level Language
Relational Algebra and Loops

Datalog

Declarative Language
Captures the Recursive Dataflow

Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler

Hyracks Dataflow

Unified Runtime
Scalability + High performance
Approach

High Level Language
Relational Algebra and Loops

ScalOps

Datalog
Declarative Language
Captures the Recursive Dataflow

Recursive Dataflow
Suite of data-parallel operators
Selected by an Optimizer / Compiler

Hyracks Dataflow
Unified Runtime
Scalability + High performance
ScalOps

• Internal Domain Specific Language (DSL)
• Written in Scala
• Relational Algebra (Filter, Join, GroupBy, …)
• Iteration support
• Rich UDF support
  • Inline Scala function calls / literals
  • Byte-code compatible with Java
• Support in major IDEs
def train(xy:Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

  class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) {
    env => {
      val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_)
      val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
def train(xy: Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

  class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_)
    val loss     = xy.map(x=>compute_loss(x,env.w)).reduce(_+_)
    env.w       -= gradient
    env.delta    = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
          compute_grad:(Example, Vector) => Vector,
          compute_loss:(Example, Vector) => Double) = {

  class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_
    val loss     = xy.map(x=>compute_loss(x,env.w)).reduce(_+_
    env.w       -= gradient
    env.delta    = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
class Example(x: Vector, y: Double)

def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
}
}

Table is our Dataset type
def train(xy: Table[Example],
           compute_grad:(Example, Vector) => Vector,
           compute_loss:(Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env => {

        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_+_)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_+_)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
    }
}
def train(xy: Table[Example],
   compute_grad:(Example, Vector) => Vector,
   compute_loss:(Example, Vector) => Double) = {

class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

loop(initialValue, (env: Env) => env.delta < eps) { env => {
   val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_
   val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_
   env.w -= gradient
   env.delta = env.lastLoss - loss
   env.lastLoss = loss
   env
}
}
}
def train(xy:Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment

val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_)  
    val loss     = xy.map(x=>compute_loss(x,env.w)).reduce(_+_)  
    env.w       -= gradient
    env.delta    = env.lastLoss - loss
    env.lastLoss = loss
    env

}
}
def train(xy:Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

  class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_
    val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
        compute_grad: (Example, Vector) => Vector,
        compute_loss: (Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env =>

      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_+_

      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_+_

      env.w -= gradient

      env.delta = env.lastLoss - loss

      env.lastLoss = loss

      env

    }

  }

}

Compute gradient

Monday, April 16, 2012
def train(xy:Table[Example],
compute_grad:(Example, Vector) => Vector,
compute_loss:(Example, Vector) => Double) = {
  class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment
  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_)
    val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
compute_grad:(Example, Vector) => Vector,
compute_loss:(Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy:Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

  class Env(w:VectorType, lastError:DoubleType, delta:DoubleType) extends Environment

  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_
    val loss      = xy.map(x=>compute_loss(x,env.w)).reduce(_+_
    env.w       -= gradient
    env.delta    = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {

val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

loop(initialValue, (env: Env) => env.delta < eps) { env => {
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
}
}
}
def train(xy: Table[Example],
  compute_grad: (Example, Vector) => Vector,
  compute_loss: (Example, Vector) => Double) {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {
    val initialValue = new Env(VectorType.zeros(10), Double.MaxValue, Double.MaxValue)
    loop(initialValue, (env: Env) => env.delta < eps) { env =>
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_+_
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_+_
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
def train(xy: Table[Example],
       compute_grad:(Example, Vector) => Vector,
       compute_loss:(Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
    loop(initialValue, (env: Env) => env.delta < eps) { env => {
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
def train(xy: Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env =>
        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
BGD in ScalOps

Shared Loop State

```scala
def train(xy: Table[Example],
        compute_grad: (Example, Vector) => Vector,
        compute_loss: (Example, Vector) => Double) = {
    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
    loop(initialValue, (env: Env) => env.delta < eps) { env =>
        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
```

Monday, April 16, 2012
def train(xy: Table[Example],
    compute_grad:(Example, Vector) => Vector,
    compute_loss:(Example, Vector) => Double) = {
    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
    loop(initialValue, (env: Env) => env.delta < eps) { env =>
        val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_)
        val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
def train(xy: Table[Example],
compute_grad: (Example, Vector) => Vector,
compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType)
  extends Environment {
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env => {
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env.lastLoss = loss
      env
    }
  }
}
def train(xy: Table[Example],
compute_grad: (Example, Vector) => Vector,
compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType,
             lastError: DoubleType,
             delta: DoubleType)
                   extends Environment {
    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) {
      env => {
        val gradient = xy.map(x=>compute_grad(x,env.w)).reduce(_+_)
        val loss = xy.map(x=>compute_loss(x,env.w)).reduce(_+_)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
      }
    }
  }
}
def train(xy: Table[Example],
compute_grad: (Example, Vector) => Vector,
compute_loss: (Example, Vector) => Double) = {

  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) {
      env => {
        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
      }
    }
  }
}
**Approach**

- **ScalOps**
  - High Level Language
  - Relational Algebra and Loops

- **Datalog**
  - Declarative Language
  - Captures the Recursive Dataflow

- **Recursive Dataflow**
  - Suite of data-parallel operators
  - Selected by an Optimizer / Compiler

- **Hyracks Dataflow**
  - Unified Runtime
  - Scalability + High performance
Approach

- ScalOps
- Relational Algebra and Loops

- Datalog
- Captures the Recursive Dataflow

- Recursive Dataflow
  - Suite of data-parallel operators
  - Selected by an Optimizer / Compiler

- Hyracks Dataflow
  - Unified Runtime
  - Scalability + High performance
def train(xy: Table[Example],
      compute_grad:(Example, Vector) => Vector,
      compute_loss:(Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env => {
      val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_+_)
      val loss = xy.map(x => compute_loss(x, env.w)).reduce(_+_)
      env.w -= gradient
      env.delta = env.lastLoss - loss
      env
    }
  }
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env => {

        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)

        env.w       -= gradient
        env.delta    = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}

Monday, April 16, 2012
def train(xy: Table[Example],
          compute_grad: (Example, Vector) => Vector,
          compute_loss: (Example, Vector) => Double) = {
    class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {

    val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)

    loop(initialValue, (env: Env) => env.delta < eps) { env =>
        val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
        val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
        env.w -= gradient
        env.delta = env.lastLoss - loss
        env.lastLoss = loss
        env
    }
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {

class Env(w: Vector, lastError: Double, delta: Double) extends Environment

val initialValue = new Env(Vector.zeros(1000), Double.MaxValue, Double.MaxValue)

loop(initialValue, (env: Env) => env.delta < eps) { env =>
  val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
  val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)
  env.w -= gradient
  env.delta = env.lastLoss - loss
  env.lastLoss = loss
  env
}
}
def train(xy: Table[Example],
    compute_grad: (Example, Vector) => Vector,
    compute_loss: (Example, Vector) => Double) = {
  class Env(w: VectorType, lastError: DoubleType, delta: DoubleType) extends Environment {
  val initialValue = new Env(VectorType.zeros(1000), Double.MaxValue, Double.MaxValue)
  loop(initialValue, (env: Env) => env.delta < eps) { env =>
    val gradient = xy.map(x => compute_grad(x, env.w)).reduce(_ + _)
    val loss = xy.map(x => compute_loss(x, env.w)).reduce(_ + _)

    env.w -= gradient
    env.delta = env.lastLoss - loss
    env.lastLoss = loss
    env
  }
}

Merge into one Operator

env.w -= gradient
env.delta = env.lastLoss - loss
env.lastLoss = loss
Logical Plan

MapReduce

Map()
Reduce()

Continue()
Loop

(Model, Performance)

Training Data

MapReduce

Aggregate Statistics

Update()
Sequential

Monday, April 16, 2012
Approach

High Level Language
Relational Algebra and Loops

Declarative Language
Captures the Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler

Unified Runtime
Scalability + High performance
Approach

ScalOps

High Level Language
Relational Algebra and Loops

Datalog

Declarative Language
Captures the Recursive Dataflow

Recursive Dataflow

Suite of data-parallel operators
Selected by an Optimizer / Compiler

Hyracks Dataflow

Unified Runtime
Scalability + High performance

Monday, April 16, 2012
Some Optimizations

• Caching, Rocking

• Scheduling: Data-Local, Iteration-Aware

• Avoid (de-)serialization
• Minimize network connections

• Pipelining
Physical Plan

Data Loading

HDFS → Cached Records

Iterative Computation

HDFS

(map)

CR

Aggregation tree (reduce)

Sequential (update)

HDFS

Driver (loop)
Physical Plan

Data Loading

HDFS → CR → Cached Records

Iterative Computation

HDFS → CR → (map) → Iteration Barrier → Aggregation tree (reduce) → Sequential (update) → HDFS

Driver (loop)
Physical Plan

Data Loading

HDFS → CR → Cached Records

Iterative Computation

HDFS → CR → model

(map) → Aggregation tree (reduce) → Sequential (update) → HDFS

Iteration Barrier

Driver (loop)

How Many?

Structure?
Fan-In
Fan-In
Fan-In: Blocking
Fan-In: Blocking

\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]
Fan-In: Time per Level

\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]
\[ t = fA \]
\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]
\[ t = fA \]

\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]
Fan-In: Total Time

\[ t = fA \]
\[ h = \log_f(N) = \frac{\ln(N)}{\ln(f)} \]

\[ t = h \times t \]
\[ = \frac{f}{\ln(f)} \ln(N) \times A \]
Fan-In: Total Time

\[ t = fA \]

Minimized for \( f = e \)

\[ t = h \times t = \frac{f}{\ln(f)} \ln(N) \times A \]

\[ \Rightarrow \frac{\ln(N)}{\ln(f)} = \frac{\ln(N)}{\ln(f)} \]
Partitioning

• Aggregation time increases logarithmically with number of machines

• Map time decreases linearly with the number of machines

• Closed form solutions available (but omitted here)
Evaluation

• As fast as
  • Vowpal Wabbit
  • Spark

• Faster than Hadoop (doh!)

• Much, much less code
Evaluation

Iteration time (seconds)

CPUs

Iteration Cost (CPU-seconds)

Monday, April 16, 2012
Evaluation

Optimizer: Cheapest

Iteration time (seconds)

CPUs

Iteration Cost (CPU-seconds)
Evaluation

Optimizer: Cheapest

Optimizer: Fastest

Iteration time (seconds)

Iteration Cost (CPU-seconds)

CPUs
Summary

- **Example Formation**
  - Use Pig

- **Modeling**
  - Hadoop (maybe not)
  - Subsampling (now)
  - Spark / Pregel (now)
  - ScalOps (as soon as we are done)